CLIPPEDIMAGE= JP02001034619A
PUB-NO: JP02001034619A
DOCUMENT-IDENTIFIER: JP 2001034619 A
TITLE: STORE AND RETRIEVAL METHOD OF XML DATA, AND XML DATA RETRIEVAL SYSTEM
PUBN-DATE: February 9, 2001
INVENTOR-INFORMATION:

NAME	COUNTRY
KANEMASA, YASUHIKO	N/A
KUBOTA, KAZUMI	N/A
ISHIKAWA, HIROSHI	N/A

INT-CL_(IPC): G06F017/30

ABSTRACT:
PROBLEM TO BE SOLVED: To make storable XML data into a data base and to make executable a complicated inquiry at a high speed.

SOLUTION: A relation data base of an XML data store means 1 includes an intermediate node table 2 which stores the intermediate node information, a link table 3 which stores the link information, a leaf node table 4 which stores the leaf nodes, an attribute table 5 which stores the attribute information, a path ID table 6 where the path IDs are made to correspond to the character strings and a label ID table 7 where the label Ids are made to correspond to the character strings. The XML data which are expressed in a tree structure are divided into nodes, and these nodes are made to correspond to the link information and stored in the tables 2-7. When the XML data are retrieved, an inquiry statement is given to an inquiry processing means 9. The means 9 executes an inquiry to track a tree structure by using index 8 and outputs a requested retrieval result.

DERWENT-ACC-NO: 2001-230893
DERWENT-WEEK: 200124

BASIC-ABSTRACT: NOVELTY - The tree structure of extensible mark-up language (XML) is divided into nodes and links. Information contained in intermediate and branch nodes along with links are collected and stored in the tables (2-4) in relational database memory (1). XML data with tree structure is searched with reference to the tables.

ADVANTAGE - Since the tree structure of XML is divided into nodes and links, the searching of XML data is done at a high speed even if the data structure is unique.

(19)日本国特許庁（ＪＰ）　(12) 公開特許公報（Ａ）　(11)特許出願公開番号

特開2001－34619

（P2001－34619A）

(43)公開日　平成13年2月9日(2001.2.9)

(51)Int.Cl.⁷	識別記号	ＦＩ		テーマコード゛(参考)
Ｇ０６Ｆ　17/30		Ｇ０６Ｆ　15/419	３２０	５Ｂ０７５
		15/403	３３０Ｂ	
			３４０Ｄ	

審査請求　未請求　請求項の数５　ＯＬ　（全 15 頁）

(21)出願番号　特願平11－203908

(22)出願日　平成11年7月16日(1999.7.16)

(71)出願人　000005223
富士通株式会社
神奈川県川崎市中原区上小田中４丁目１番1号

(72)発明者　金政　泰彦
神奈川県川崎市中原区上小田中４丁目１番1号　富士通株式会社内

(72)発明者　久保田　和己
神奈川県川崎市中原区上小田中４丁目１番1号　富士通株式会社内

(74)代理人　100100930
弁理士　長澤　俊一郎　（外１名）

最終頁に続く

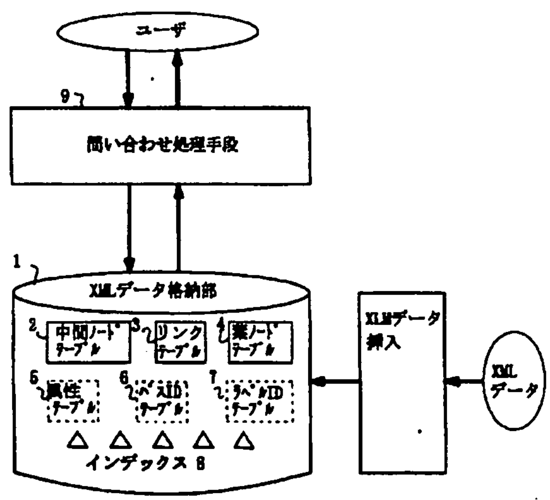
(54)【発明の名称】　ＸＭＬデータの格納／検索方法およびＸＭＬデータ検索システム

(57)【要約】

【課題】　ＸＭＬデータをデータベースに格納し、複雑な問合わせを高速に実行すできるようにすること。

【解決手段】　ＸＭＬデータ格納手段１の関係データベースに、中間ノードの情報を格納する中間ノードテーブル２、リンクの情報を格納するりンクテーブル３、葉ノードの情報を格納する葉ノードテーブル４、属性情報を格納する属性テーブル５、パスＩＤと文字列とを対応付けたパスＩＤテーブル６、ラベルＩＤと文字列を対応表づけたラベルＩＤテーブル７を設け、木構造で表現されたＸＭＬデータをノード単位で分割し、上記テーブル２～７に各ノードとリンク情報を関係付けて格納する．ＸＭＬデータを検索するには、問い合わせ処理手段９に対し問い合わせ文により問い合わせを行う．問い合わせ処理手段９は、インデックス８を用いて木構造を辿る問い合わせを実行し、要求された検索結果を出力する．

本発明の基本構成図



【特許請求の範囲】

【請求項1】　XMLで記述されたデータを、エレメントを中間ノードとし、エレメント値と属性値を葉ノードとし、タグをリンクとする木構造で表現し、
XMLの木構造をノードとリンクに分解し、各ノードとリンク情報を関係付けて関係データベースのテーブルに格納し、
上記関係データベースに格納されたテーブルを利用して、任意の構造のXMLデータを検索することを特徴とするXMLデータの格納／検索方法。

【請求項2】　エレメントを中間ノードとし、エレメント値と属性値を葉ノードとし、タグをリンクとする木構造で表現されるXMLで記述されたデータを検索するシステムであって、
上記システムは、XMLデータを格納する格納手段を備え、該格納手段の関係データベースに、少なくとも中間ノードの情報を格納するための中間ノードテーブルと、リンクの情報を格納するためのリンクテーブルと、葉ノードの情報を格納するための葉ノードテーブルとを設け、
上記XMLの木構造をノードとリンクに分解して、上記テーブルに各ノードとリンク情報を関係付けて格納し、
上記テーブルを参照して木構造を辿る問い合わせを実行し、XMLデータを検索することを特徴とするXMLデータ検索システム。

【請求項3】　関係データベースに、パスの文字列とパス用のIDの対応表であるパスIDテーブルと、ラベルの文字列とラベル用IDの対応表であるラベルIDテーブルとを設けたことを特徴とする請求項2のXMLデータ検索システム。

【請求項4】　リンクテーブルの中に各子エレメントがそのエレメント内で出現した順序の情報を付加し、葉ノードテーブルの中に各エレメント値がそのエレメント内で出現した順序の情報を付加し、上記情報により元のXML文書の復元を可能としたことを特徴とする請求項2または請求項3のXMLデータ検索システム。

【請求項5】　中間ノードテーブルに、ノードIDによる検索を高速に行なうためのインデックスと、テーブルの文書IDによる検索を高速に行なうためのインデックスと、パスIDによる検索を高速に行なうためのインデックスを用意し、
リンクテーブルに、親ノードから子ノードを高速に検索するためのインデックスと、子ノードから親ノードを高速に検索するためのインデックスを用意し、
葉ノードテーブルに、ノードIDからそのノードの値を得るためのインデックスと、ある値を持つノードを検索するためのインデックスを用意し、
パスIDテーブルに、パスの文字列に対応するパスIDを検索するためのインデックスを用意し、
ラベルIDテーブルに、ラベルの文字列に対応するラベルIDを検索するためのインデックスを用意し、
上記インデックスを用いて木構造を辿る問い合わせを実行することを特徴とする請求項2，3または請求項4のXMLデータ検索システム。

【発明の詳細な説明】

【0001】

【発明の属する技術分野】本発明は、XMLで記述された大量のデータを関係データベースに格納し、検索するXMLデータの格納／検索方法および検索システムに関し、特に、XML文書の構造に依存せずにあらゆるXMLデータを格納できるようにし、また格納されたXMLデータに対するXMLの木構造を辿る問い合わせを高速に実行できるようにしたXMLデータの格納／検索方法および検索システムに関するものである。

【0002】

【従来の技術】現在、XMLデータを格納するのに用いられている手法は、大まかに次の2つのタイプに分類することができる。

①ファイル格納：XML文書をファイル形式のまま格納する手法。この手法は、オリジナルのXMLファイルの全体あるいは一部をそのまま利用することを目的としており、そのため、XML文書をファイル形式のまま格納する。しかし、それだけでは、ファイルの数が増えたときに目的とするファイルを見つけ出すことが困難になるので、目的とするファイルを検索する為のインデックスも用意しておく必要がある。

【0003】②テーブル格納：XMLを関係データベースのテーブルにマッピングして格納する手法。この手法ではXML文書を構造化データと見なし、データベースに格納することによって高速な検索を行なうことを目的としている。そのため、この手法では、各エレメントを関係データベースのテーブルの各カラムにマッピングして格納する。XMLデータをテーブルにマッピングする為には、XMLの各エレメントをテーブルの各カラムにどのようにマッピングするかというマッピング規則が必要である。このマッピング規則はユーザが事前に指定する必要がある。

【0004】

【発明が解決しようとする課題】XMLデータを格納する際に一番問題となるのは、そのデータ構造が一意に定まっていないという事である。特に、DTD（文書型宣言）のないXMLデータでは、どこにどのようなタグが出現するか分からず、データ構造は全く分からない。DTDのあるXMLデータでさえも、DTDの中でタグの繰り返しやタグの選択、タグの再帰的な宣言が許されているので、データ構造が一意に定まらない。なお、このようなデータを半構造データと呼ぶ。このようなデータ構造の定まっていないXMLデータを格納しようとすると、格納スキーマの設計が問題となる。例えば、図8に示される〔DTD〕を持つ、サンプルXMLデータ〔X

MLデータ〕をテーブル格納でデータベースに格納した場合を考える。なお、このサンプルXMLデータは、２冊の本の情報を含む書籍目録のデータである。

【０００５】図9は上記XMLデータをテーブルに格納した様子を示す図である。図9のテーブルでは、１タプルが本１冊分の情報に相当していて、列にはXMLデータ中で出現する可能性のある全てのタグがとられている。これを見ると、一見サンプルデータが問題なく格納されているかのように見える。しかし、サンプルデータのＤＴＤに書かれた定義には著者数の制限が無いのに、図9のテーブルでは著者を格納するスペースは最大２人分しか用意されていない。もしXMLデータの中に著者がそれ以上存在したら、そのデータは格納できないか、格納しても情報が一部欠損することになる。このように、テーブル格納では、XMLのＤＴＤで記述される繰り返しタグを格納することができない。これは、テーブル格納ではあらかじめ格納する要素を列として指定しておく必要があるので、最大数が未定の繰り返し要素を表現できないからである。また、同じ理由で再帰的に定義されているタグも格納できない。さらに、そもそもXMLデータにＤＴＤが存在しなくて、どのようなタグが出現するか分かっていないときには、テーブルの構造を決められず、全く対応できない。

【０００６】一方、ファイル格納は、XMLデータをファイル形式のまま格納するので、ＤＴＤの無いXMLデータであろうと半構造のXMLデータであろうと、格納できないXMLデータは存在しない。しかし、それだけでは大量に格納されたデータの中から自分の求める情報だけを検索することができないので、検索用のインデックスが必要となる。インデックスの構成は目的に応じて色々と考えられ、簡単なものではタグ名と文字列の組をキーにして、そのタグに囲まれてその文字列が出現しているようなXML文書を検索してくるというものがある。しかし、そのような簡単なインデックスでは、タグの階層構造を考慮した検索は行なえない。タグの階層構造の情報を持つようにインデックスを工夫することも考えられるが、それでもなお次のことが問題として残る。

【０００７】① インデックスがXMLの木構造の全ての情報を持っていないので、XMLデータの全情報を使った検索ができない。

② インデックスが木構造を辿ることに最適化されていないので、そのような検索を行なった場合は検索速度が遅い。

以上のように、データ構造が一意に定まっていないXMLデータにおいては、いかにしてＤＴＤ無しのXMLデータや半構造のXMLデータを格納するか、また、格納されたXMLデータに対していかにして木構造を辿るような複雑な問い合わせを高速に実行できるようにするかといった問題がある。本発明は上記した事情に鑑みなされたものであって、本発明の目的は、データ構造が一意に定まっていないXMLデータをデータベースに格納し、複雑な間合わせを高速に実行することができるXMLデータの格納／検索方法およびXMLデータ検索システムを提供することである。

【０００８】

【課題を解決するための手段】図１は本発明の基本構成を示す図である。同図に示すように、本発明のシステムは、エレメントを中間ノードとし、エレメント値と属性値を葉ノードとし、タグをリンクとする木構造で表現されるXMLで記述されたデータを検索するシステムにおいて、XMLデータを格納する格納手段１を設け、該格納手段１の関係データベースに、少なくとも中間ノードの情報を格納するための中間ノードテーブル２と、リンクの情報を格納するためのリンクテーブル３と、葉ノードの情報を格納するための葉ノードテーブル４とを設ける。そして、上記XMLの木構造で表現されたXMLデータをノード単位で分割し、上記テーブル２～４に各ノードとリンク情報を関係付けて格納する。XMLでは、木構造を形成する中間ノードと、エレメントの値を持っている葉ノードとでは、格納するために最適な格納構造が異なるので、上記のようにそれぞれ最適化された別々の専用テーブルに格納するのが望ましい。このように、値を持つためのノードである葉ノードと木構造の情報を持つためのノードである中間ノードを別々のテーブルに格納することにより、値を格納するための格納スペースを節約することが可能となる。各ノード間の接続情報を保持する為のリンクも、リンクテーブル３に格納して持っておく必要がある。また、属性情報を格納するための属性テーブル５を別途設けてもよい。さらに、中間ノードテーブル２に各ノードのルートからのフルパス情報をＩＤで記述し、パス用のＩＤと文字列の対応表をパスＩＤテーブル６として別に持つことにより、格納スペースの節約と、検索の高速化を図ることができる。同様に、リンクテーブル３のタグ名と属性ノードテーブルの属性名をＩＤで記述し、これらラベルのＩＤと文字列の対応表をラベルＩＤテーブル７として別に持つことによって、格納スペースの節約と文字列検索の高速化を図ることができる。また、リンクテーブル３の中に各子エレメントがそのエレメント内で出現した順序の情報を付加し、葉ノードテーブルの中に各エレメント値がそのエレメント内で出現した順序の情報を付加することにより、元のXML文書の復元が可能となる。

【０００９】本発明では、XMLの木構造をそのまま格納手段１に格納するので、ＤＴＤ無しのXMLデータや半構造のXMLデータも格納できる。また、XMLの木構造を全てデータベース上に格納しているので、木構造の全ての情報を検索に利用することができる。しかしこれだけでは問い合わせが行なわれたときに、ノード単位に分割して格納されているXMLデータの木構造を再結合するのに時間がかかり、問い合わせの実行時間が遅く

なる。そこで本発明では、上記のテーブル２～７に、ＸＭＬデータへの問い合わせパターンを考慮してインデックス８を張る。これにより、ＸＭＬの木構造を辿るような複雑な問い合わせの実行を高速に行なうことを可能となる。上記ＸＭＬデータを検索するには、例えばＸＭＬデータ検索言語により、問い合わせを行う。これにより問い合わせ処理手段９は、問い合わせ文の構文チェックを行い問い合わせのための構文木を生成し、最適な実行プランを生成する。この実行プランは、木構造検索用の関数セットで記述される。この実行プランにより、上記インデックス８を用いて木構造を辿る問い合わせを実行し、要求された検索結果を出力する。

【００１０】本発明においては、次のように構成することもできる。

（１）テーブルに関係データベースの制約の機能を適用することによって、ＸＭＬの構文規則をチェックする。

（２）リンクテーブルの中に、各エレメントの同ラベルを持つ兄弟エレメント中での出現順序の情報を付加し、各ラベルの出現順序を指定した問い合わせの実行を可能とする。

（３）リンクテーブルにリンクの両節点の情報だけでなくタグ名の情報も待つことによって、タグ名を指定してリンクを辿る問い合わせを高速に実行する。

（４）属性テーブルの中の属性ノードの接続先をリンクではなくて中間ノードこすることによって、属性を条件にして木構造を辿る問い合わせを実行する際のテーブル検索回数を削減し、問い合わせの高速実行を可能とする。

（５）中間ノードテーブルのパスＩＤによる検索を高速に行なうためのインデックスをB⁺ -tree で構築する場合において、キー値をパスＩＤとノードＩＤの組とすることによってキー値の重複を無くす。

（６）中間ノードテーブルの文書ＩＤによる検索を高速に行なうためのインデックスをB⁺ -tree で構築する場合において、キー値を文書ＩＤとノードＩＤの組とすることによってキー値の重複を無くす。

【００１１】

【発明の実施の形態】以下、本発明の実施の形態について説明する。

（１）システム構成

図２は本発明の実施例のシステムの構成を示す図である。同図に示すように、本実施例のシステムは大きくわけて、ＸＭＬデータ格納部１１、ＸＭＬデータ格納部１１にＸＭＬデータを挿入するためのＸＭＬデータ挿入モジュール１２、格納されたＸＭＬデータへの問い合わせを処理する問い合わせ処理エンジン部１３から構成される。ＸＭＬデータは、ＸＭＬデータ挿入モジュール１２によって、ＸＭＬデータ格納部１１に挿入される。ＸＭＬデータ挿入モジュール１２は、ＸＭＬパーザ１２ａとローダー１２ｂから成り、ＸＭＬパーザ１２ａは入力されたＸＭＬデータを構文解析し、ＸＭＬデータの木構造を、ＸＭＬデータ格納部１１に格納できるようにノード単位に分解する。また、ローダー１２ｂは、そのノード単位に分解された木構造をＸＭＬデータ格納部１１のテーブルに挿入する。

【００１２】図３に上記ＸＭＬデータの格納処理を示すフローチャートを示す。本実施例においてＸＭＬデータの格納処理は次のように行われる。まず、ステップＳ１において、ＸＭＬファイルを読み込む。ステップＳ２において、ＸＭＬパーザにより、入力ファイルの構文解析を行う。解析が成功した場合には、ステップＳ３に行き、ＸＭＬパーザが解析結果として、ＸＭＬの木構造のノード情報とリンク情報を中間形式としてファイル出力する。また、解析が成功しない場合には、構文解析失敗としてエラー出力し処理を終了する。ステップＳ４において、生成された中間形式ファイルを読み込み、ステップＳ５において、読み込んだＸＭＬデータをローダによって関係データベースの各テーブルに挿入し、処理を終了する。また、上記挿入が成功しない場合には、データ挿入失敗としてエラー出力をして処理を終了する。

【００１３】格納されたＸＭＬデータに対する問い合わせは、ＸＭＬデータ問い合わせ言語で行なわれ、その問い合わせは問い合わせ処理エンジン１３で処理される。問い合わせ処理エンジン１３は、問い合わせ言語のパーザ１３ａ、問い合わせ最適化エンジン１３ｂ、木構造検索用ＡＰＩ（アプリケーション・プログラミング・インタフェース）１３ｃから成る。問い合わせ言語のパーザ１３ａは、入力された問い合わせ文の構文チェックを行い問い合わせのための構文木を生成する。問い合わせ最適化エンジン１３ｂは、上記構文木を基に、最適な実行プランを生成する。この実行プランは、木構造検索用ＡＰＩ１３ｃの関数セットで記述される。木構造検索用ＡＰＩ１３ｃは、ＸＭＬデータ格納部１１とのインタフェースで、ＸＭＬの木構造上での基本的な検索を行なう関数のセットである。

【００１４】次に、上記システムにおける各部の構成についてさらに詳細に説明する。

（１）テーブル構成

まず、上記ＸＭＬデータ格納部１１に格納されるテーブルの構成について説明する。ＸＭＬデータを木構造で表現する方法はいくつかあるが、本実施例では図４に示す木構造表現を想定している。図４は、前記図８に示したＸＭＬデータを木構造で表現したものである。この木構造表現において、丸い中間ノードはエレメントを表しており、ノードの親子関係がエレメントの包含関係を表している。

【００１５】また、ノードの丸の中の数字はノードＩＤを表している。ノードとノードを結ぶリンク（枝）はタグを表しており、リンクの横に書かれている文字列はタグ名を表している。三角の葉ノードはエレメントの値を

表し、四角い葉ノードはタグに付けられた属性(Atrribute)を表している。値を持つのはこの２つの葉ノードだけである。ノードを分割してデータベースに格納するときに、ノードの情報だけをデータベースのテーブルに格納したのでは、木構造のノード間の繋がり、つまりリンクの情報が欠落してしまう。そこで、リンクの情報はリンクの情報としてそれを格納する専用のテーブルを用意する。またノードも、中間ノードと、エレメント値の葉ノード、属性の葉ノードとは最適な格納構造が異なるので、別々のテーブルに格納する必要がある。

【００１６】本実施例で使用するテーブルは、全部で次の6つである。

①中間ノードテーブル

これは中間ノードの情報を格納するテーブルである。ノードＩＤ(id)の他に、そのノードが含まれている文書の文書ＩＤ(docid)、そのノードまでのルートからのフルパスのＩＤ(pathid)をカラムとして持っている。

②リンクテーブル

これはノード間のリンクを格納するテーブルである。ノードＩＤ(id)、リンクのラベル（タグ名）のＩＤ(labelid)、子ノードのノードＩＤ(child)、その子ノードの全兄弟ノード中での出現順序(tord:total order)、その子ノードの同ラベルを持つ兄弟ノード中での出現順序(pord:partial order)をカラムとして持っている。上記のように、リンクテーブル中にラベル（タグ名）のＩＤ(labelid)を付加することによりタグ名を指定してリンクを辿る問い合わせを高速に実行することが可能となる。

【００１７】③葉ノードテーブル

これはエレメント値の葉ノードを格納するテーブルである。そのエレメントにあたる中間ノードのノードＩＤ(id)の他に、エレメントの値(value)と、そのエレメント中でその値が出現した順序(order)をカラムとして持っている。このように、値を持つための葉ノードテーブルを、前記中間ノードテーブルとは別に設けることにより、値を格納するスペースを節約することができる。

【００１８】④属性ノードテーブル

これはタグにつけられた属性（例えば図８における<book year="1995">におけるyear）を格納するテーブルである。そのタグが含まれるエレメントにあたる中間ノードのノードＩＤ(id)の他に、属性名のＩＤ(labelid)、属性値(Attvalue)をカラムとして持つ。なお、属性テーブルに関係データベースの制約機能を用いて、(id,labelid)の組がユニークという制約をかけておくことによって、「同一のタグ内では同一の属性名は出現してはならない」というＸＭＬの属性に関する構文規則をチェックすることができる。また、本実施例で想定している木構造表現では、ＸＭＬのタグが木構造のリンクに相当するので、ＸＭＬのタグに付けられる属性は本来ならばリンクに付くべきである。しかし、図4では、属性はリンクに対してではなく、その下のノードに付いている。これは、検索時のテーブル参照の回数を少なくするためである。すなわち、属性を条件として木構造を辿る問い合わせを実行する際のテーブル検索回数を削減し、問い合わせの高速化を図ることが可能となる。

【００１９】⑤パスＩＤテーブル

これはパスＩＤとパスの文字列の対応表である。パスの文字列を中間ノードテーブルに直接書き込まないでこのように別に持っているのは、スペースの節約の為もあるが、パス名の文字列マッチングを含む検索が行なわれたときに、検索対象が少なくてすみ、検索が高速化できるからでもある。

⑥ラベルＩＤテーブル

これはラベルＩＤとラベルの文字列の対応表である。このように、リンクテーブルのタグ名と、属性ノードテーブルの属性名をＩＤで記述し、このラベルのＩＤと文字列の対応表をラベルＩＤテーブルとして別に持つことにより、パスＩＤテーブルと同様、格納スペースの節約と、検索の高速化を図ることができる。

【００２０】また、上記のように、リンクテーブル中に、子ノードの全兄弟ノード中での出現順序(tord:total order)の情報を付加し、また、葉ノードテーブル中に、各エレメント値がそのエレメント内で出現した順序(order)の情報を付加することに、ＸＭＬデータ格納部１１に格納されるノード単位に分解されたＸＭＬデータから、元のＸＭＬ文書を復元することが可能となる。例えば、「今日は <天気> 晴れ</天気> だった。〇〇は <場所> デパート</場所> へでかけた。」のようにタグで区切られた文章を復元することも可能になる。また、リンクテーブル中に、各エレメントの同ラベルを持つ兄弟ノード中での出現順序(pord:partial order)の情報を付加することにより、各ラベルの出現順序を指定した問い合わせを高速に実行することが可能となる。

【００２１】一例として、図8のサンプルＸＭＬデータ（図４の木構造表現）を上記のテーブル群で格納した様子を図５、図６に示す。図5は中間ノードテーブル、リンクテーブルの例を示す図である。中間ノードテーブルにおいて、例えば、第１行目のid（＝5）は図4において”５”と記されたノードを示し、そのノードが含まれている文書の文書ＩＤ(docid)は1である。また、そのノードまでのルートからのフルパスのＩＤ(pathid)は1であり、このＩＤに対応したpathは、"bib.book.publisher.name"である。また、リンクテーブルにおいて、例えば１行目のid（＝４）は図4において、”４”と記されたノードを示し、そのlabelidは5であり、このlabelid に対応するlabel は"name"である。また、その出現順序を示すtord,pord はそれぞれ”０”,”０”であり、子ノードは、図4で”５”と記されたノードである。

【００２２】図6は葉ノードテーブル、属性ノードテーブル、パスＩＤテーブル、ラベルＩＤテーブルの例を示す図である。葉ノードテーブルにおいて、例えば第１行

目のid（＝5）は図4において、"5" と記されたノードを示し、そのorder は" 0"、またその葉ノードの値(value)は"Addison-Wesley"である。属性ノードテーブルにおいて、例えば第1行目のid（＝3）は図4において、" 3" と記されたノードを示し、そのlabelid は3（"year"に対応）、その属性値（attvalue）は "1995" である。また、パスIDテーブル、ラベルIDテーブルにはそれぞれ、上記各テーブル中のpathid、labelid に対応したパスの文字列、ラベルの文字列が格納され、例えば、pathid＝" 1" に対応した文字列は前記したように" bib.book.publisher.name"であり、また、例えばlabelid ＝" 1" に対応した文字列は" bib"である。

【0023】（2）インデックスの構成
本実施例においては、本来連結されていたはずの木構造のノードが、前記したように1つ1つに分割されて関係データベースのテーブルに格納されている。このために、木構造を辿る問い合わせが行なわれた場合、問い合わせで辿る部分のリンクを連結し直すためにジョイン操作が行なわれる。このジョイン操作の速度は全体の検索速度に大きく影響するので、ジョイン操作を高速に行なえるようにインデックスを効果的に張っておく必要がある。また、問い合わせが行なわれる場合、検索条件として指定されるのは、エレメントの値、属性、パス、出現順序などである。それらの検索も高速に行なう必要があるので、そこにもインデックスを用意しておく必要がある。

【0024】図7に、上記図5、図6に示したテーブルに張ったインデックスの一覧を示す。このインデックスは B⁺ -tree で張ってあり、キーが複数の属性の組からなるインデックスは、その組の先頭からの部分的な属性の組で検索に用いることもできる。なお中間ノードテーブルに張ってあるインデックスでキーが(pathid,id) のものは、あるパスに該当する全てのノードを検索してくるときに使用するものである。このインデックスのキーは、一見pathid単独で構わないように思われるかもしれない。しかしキーをpathidだけにすると、同じキー値を持つエントリが多量に発生して、B⁺ -tree インデックスが機能しなくなる。上記のようにキー値をパスID(pathid)とノードのID(id)の組とすることにより、キー値の重複を無くすことができ、B⁺ -tree の検索を高速に行うことができる。また、中間ノードテーブルに張ってあるインデックスでキーが(docid,id)も同様であり、文書ID(docid) とノードのID(id)の組とすることにより、キー値の重複を無くすことができ、B⁺ -tree の検索を高速に行うことができる。

【0025】（3）問い合わせの実行
前記したように、格納されたXMLデータに対する問い合わせは、例えばXMLデータの問い合わせ言語で行なわれる。XMLデータのための検索言語の一つとして検索言語XQLがある。XQLによる問い合わせ文を、例により簡単に説明する。

【0026】

```
SELECT result:<$book.title>
FROM  book: bib.book
WHERE $book.author.lastname="Darwen";
```

この問い合わせの意味は「bib.book.author.lastnameがDarwenであるようなbib.bookについて、bib.book.title を検索結果として得たい」という意味である。

【0027】上記に示すように、問い合わせ文は大きく、SELECT、FROM、WHERE の3つの部分に別れている。SELECTの部分では検索結果として得たいエレメントのプロジェクションを指定する。FROMの部分では検索の対象となるエレメントを指定している。WHERE の部分では検索の条件のセレクションを指定する。上記のような問い合わせは前記したように、問い合わせ処理エンジン13で処理される。問い合わせ処理エンジン13では、上記のような問い合わせ文の構文チェックを行い問い合わせのための構文木を生成する。そして、該構文木を基に、最適な実行プランを生成する。この実行プランは、木構造検索用の関数セットで記述される。

【0028】次に、上記XMLデータに対する問い合わせ処理が、どのように行なわれるかを説明する。ここでは、図8のサンプルXMLデータを、XMLデータ格納部11に格納し、前述した図5、図6に示したテーブルに挿入した場合を例として、上記のように「著者がDarwenである本のタイトルを求めよ」という問い合わせを行なった場合について説明する。この場合のテーブル検索は、次のように行われる。なお、下記1．～10．の処理は、上記木構造検索用の関数により実行される。

【0029】1．葉ノードテーブルを検索して、値が "Darwen" であるノードのノードID（＝16）を得る。
2．パスIDテーブルを検索して、パス"bib.book.author.lastname " のパスID（＝4）を得る。
3．中間ノードテーブルを上記1．で得られたノードID（＝16）で検索して、得られたパスID（＝4）が上記2．で得られたパスID（＝4）と一致することを確認する。
4．ラベルIDテーブルを検索して、ラベル"lastname"のラベルID（＝8）を得る。
5．リンクテーブルを検索して、上記1．で得られたノードID（＝16）と上記4．で得られたラベルID（＝8）から、親ノードのノードID（＝15）を得る。
6．ラベルIDテーブルを検索して、ラベル" author " のラベルID（＝7）を得る。
7．リンクテーブルを検索して、上記5．で得られたノードID（＝15）と上記6．で得られたラベルID（＝7）から、親ノードのノードID（＝9）を得る。

8．ラベルＩＤテーブルを検索して、ラベル"title" のラベルＩＤ（＝6）を得る。
9．リンクテーブルを検索して上記7．で得られたノードＩＤ（＝9）と上記8．で得られたラベルＩＤ（＝6）から、子ノードのノードＩＤ（＝１２）を得る。
１０．葉ノードテーブルを検索して、上記9．で得られたノードＩＤ（＝１２)から、そのノードの値("Foundation for Object/Relational Database")を得る。以上のようにして得られた検索結果は、問い合わせ処理エンジン１３を介して出力され、ユーザに提示される。

【００３０】

【発明の効果】以上説明したように、本発明においては、関係データベースに、中間ノードの情報を格納するための中間ノードテーブルと、リンクの情報を格納するためのリンクテーブルと、葉ノードの情報を格納するための葉ノードテーブル等のテーブルを設け、ＸＭＬの木構造をノードとリンクに分解して、上記テーブルに各ノードとリンク情報を関係付けて格納し、上記テーブルを参照して木構造を辿る問い合わせを実行し、ＸＭＬデータを検索するようにしたので、データ構造が一意に定まっていないＸＭＬデータに対する複雑な問い合わせを高速に実行することができる。また、ＸＭＬの木構造をそのまま格納手段に格納するので、ＤＴＤ無しのＸＭＬデータや半構造のＸＭＬデータも格納することができる。さらにＸＭＬの木構造を全てデータベース上に格納しているので、木構造の全ての情報を検索に利用することができる。

【図面の簡単な説明】

【図１】本発明の基本構成図である。

【図２】本発明の実施例のシステムの構成例を示す図である。

【図３】本発明の実施例のシステムにおける格納処理フローを示す図である。

【図４】ＸＭＬデータの木構造表現の一例を示す図である。

【図５】本発明の実施例のテーブル構成の一例を示す図（1）である。

【図６】本発明の実施例のテーブル構成の一例を示す図（2）である。

【図７】本発明の実施例のイッデックス一覧を示す図である。

【図８】ＸＭＬデータの一例を示す図である。

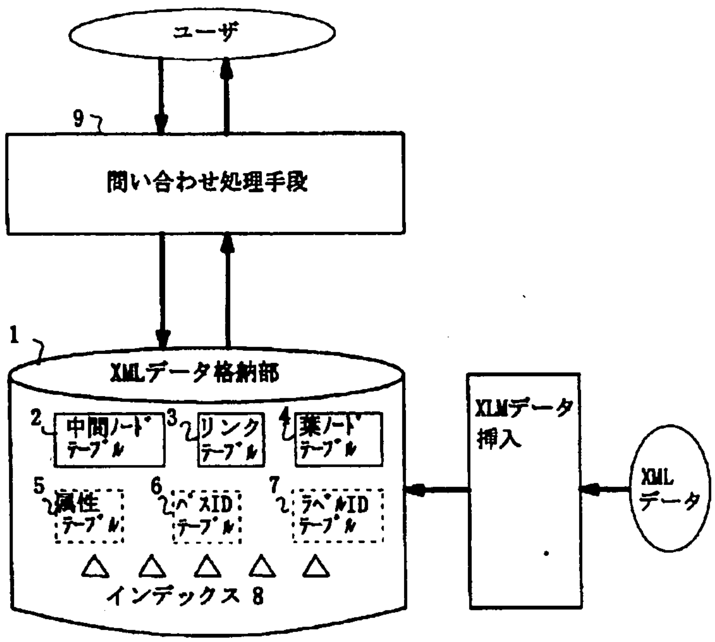
【図９】図8のＸＭＬデータをテーブルに格納した様子を示す図である。

【符号の説明】

1	ＸＭＬデータ格納格納手段
2	中間ノードテーブル
3	リンクテーブル
4	葉ノードテーブル
5	属性テーブル
6	パスＩＤテーブル
7	ラベルＩＤテーブル
8	インデックス
9	問い合わせ処理手段
１１	ＸＭＬデータ格納部
１２	ＸＭＬデータ挿入モジュール
１２ａ	ＸＭＬパーザ
１２ｂ	ローダ
１３	問い合わせ処理エンジン部
１３ａ	問い合わせ言語のパーザ
１３ｂ	問い合わせ最適化エンジン
１３c	木構造検索用

【図1】

本発明の基本構成図



【図7】

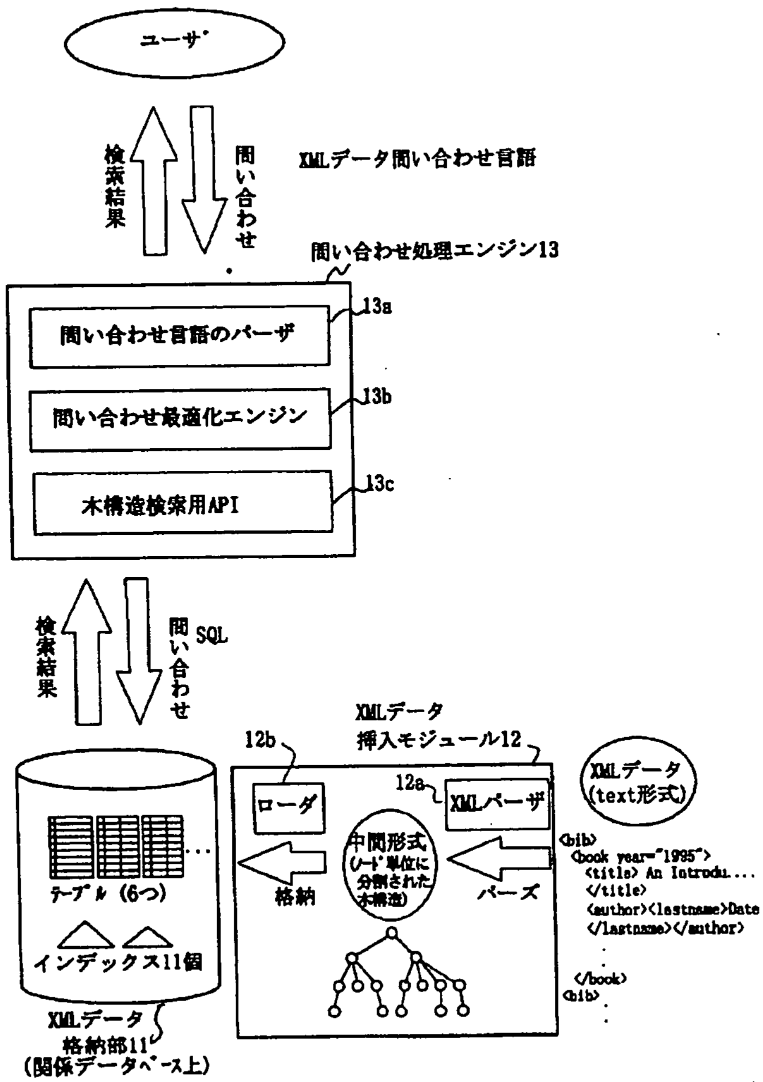
本発明の実施例のイッデックス一覧を示す図

インデックス一覧

テーブル名	キー
中間ノード	id
中間ノード	(docid, id)
中間ノード	(pathid, id)
リンク	(id, labelid, child)
リンク	(child, labelid, id)
葉ノード	id
葉ノード	value
属性ノード	id
属性ノード	attvalue
パスＩＤ	path
ラベルＩＤ	label

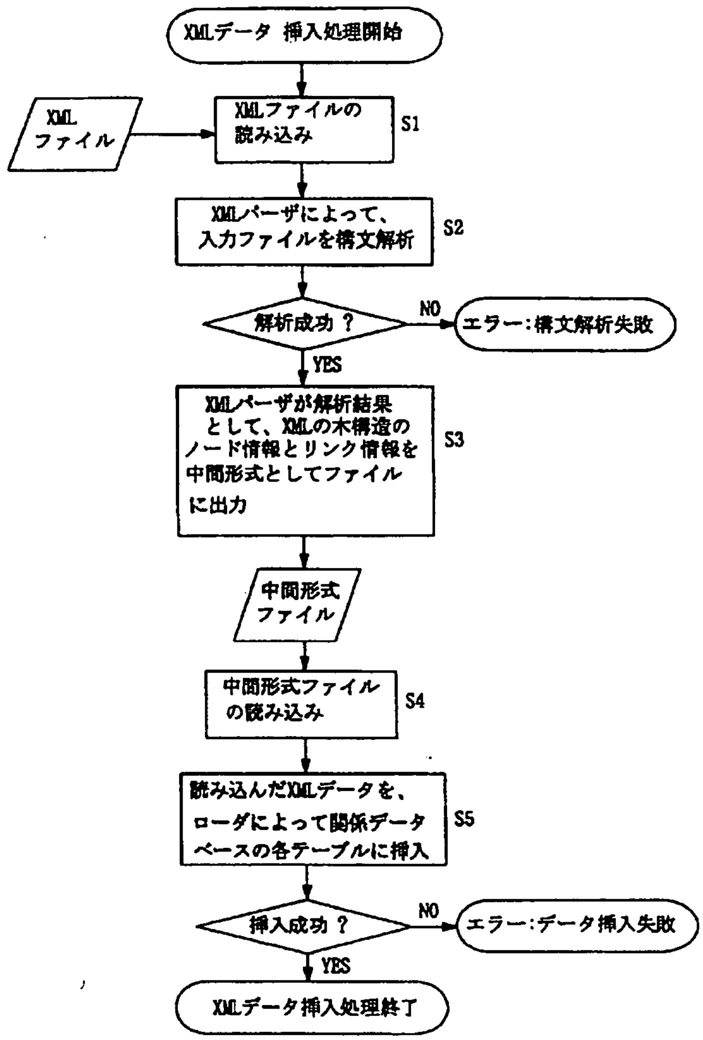
【図2】

本発明の実施例のシステムの構成例を示す図



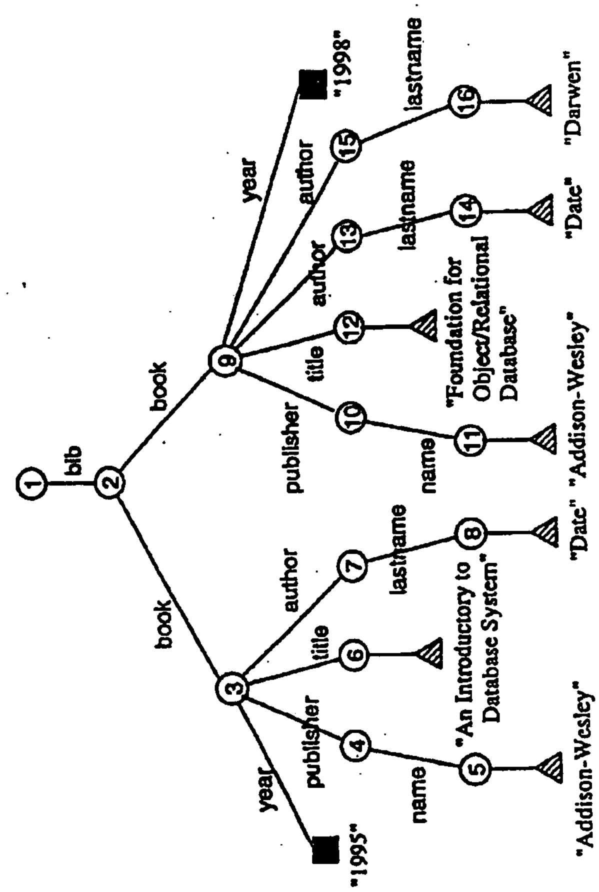
【図3】

本発明の実施例のシステムにおける格納処理フローを示す図



【図4】

XMLデータの木構造表現の一例を示す図



【図5】

本発明の実施例のテーブル構成の一例を示す図（1）

中間ノードテーブル

id	docid	pathid
5	1	1
4	2	2
6	1	3
8	1	4
7	1	5
3	1	6
11	1	1
10	1	2
12	1	3
14	1	4
13	1	5
16	1	4
15	1	5
9	1	6
2	1	7
1	1	8

リンクテーブル

id	labelid	tord	pord	child
4	5	0	0	5
7	8	0	0	8
3	4	0	0	4
3	6	1	0	6
3	7	2	0	7
10	5	0	0	11
13	8	0	0	14
15	8	0	0	16
9	4	0	0	10
9	6	1	0	12
9	7	2	0	13
9	7	3	1	15
2	2	0	0	3
2	2	1	1	9
1	1	0	0	2

【図6】

本発明の実施例のテーブル構成の一例を示す図（2）

葉ノードテーブル

id	order	value
5	0	Addison-Wesley
6	0	An Introductory to Database System
8	0	Date
11	0	Addison-Wesley
12	0	Foundation for Object/Relational Database
14	0	Date
16	0	Darwen

属性ノードテーブル

id	labelid	attvalue
3	3	1995
9	3	1998

パスIDテーブル

pathid	path
1	bib.book.publisher.name
2	bib.book.publisher
3	bib.book.title
4	bib.book.author.lastname
5	bib.book.author
6	bib.book
7	bib
8	/

ラベルIDテーブル

labelid	label
1	bib
2	book
3	year
4	publisher
5	name
6	title
7	a uthor
8	lastname

【図8】

XMLデータの一例を示す図

[DTD]

```
<!ELEMENT book (author+, title, publisher)>
<!ATTLIST book year CDATA>
<!ELEMENT article (author+, title, year?, (shortversion | longversion))>
<!ATTLIST article type CDATA>
<!ELEMENT publisher (name, address?)>
<!ELEMENT author (firstname?, lastname)>
```

[XMLデータ]

```
<bib>
    <book year="1995">
        <title> An Introductory to Database System </title>
        <author> <lastname> Date </lastname> </author>
        <publisher> <name> Addison-Wesley </name> </publisher>
    </book>

    <book year="1998">
        <title> Foundation for Object/Relational Database </title>
        <author> <lastname> Date </lastname> </author>
        <author> <lastname> Darwen </lastname> </author>
        <publisher> <name> Addison-Wesley </name> </publisher>
    </book>
</bib>
```

【図9】

図８のXMLデータをテーブルに格納した様子を示す図

ｂｏｏｋのテーブル

ID	title	author1 firstname	author1 lastname
1	An Introductory to Database System		Date
2	Foundation for Object/Relational Database		Date
⋮	⋮	⋮	⋮

author2 firstname	author2 lastname	publisher name	publisher address	year
		Addison-Wesley		1995
	Darwen	Addison-Wesley		1998
⋮	⋮	⋮	⋮	⋮

フロントページの続き

(72)発明者　石川　博
神奈川県川崎市中原区上小田中4丁目1番
1号　富士通株式会社内

Ｆターム(参考)　5B075 ND36 PP23 QR00 QT06

01348571 SUPPLIER NUMBER: 08156508 (THIS IS THE FULL TEXT)

Self-adjusting data structures: use self-adjusting heuristics to improve the performance of your applications. (technical)

Liao, Andrew M.

Dr. Dobb's Journal, v15, n2, p44(12)
Feb, 1990
DOCUMENT TYPE: technical
ISSN: 1044-789X
LANGUAGE: ENGLISH
RECORD TYPE: FULLTEXT; ABSTRACT
WORD COUNT: 4770
LINE COUNT: 00361

ABSTRACT: Structural self-adjusting heuristic' algorithms are developed that improve the performance of operations executed on such data structures as binary search trees , priority queues (heaps), and lists. A self-adjusting algorithm is implemented as a move-to-front approach for singly linked lists, which are groups of records containing fields for individual pieces of user data and pointers to succeeding records on a list. The two types of self-adjusting restructuring heuristics for a binary tree include a 'bottom-up splay' version and a 'top-down splay' version. Structural self-adjusting heuristics for heaps create a 'top-down-skew heap' self-adjusting analog of a conventional leftist heap and a 'pairing heap' as a self-adjusting analog of a 'binomial heap.' Conversion, functioning, and coding of the self-adjusting data structures are described.

TEXT:

Self-Adjusting Data Structures

Application programs are often developed using standard data structure techniques such as stacks, queues, and balanced trees with the goal of limiting worst case performance. Such programs, however, normally carry out many operations on a given data structure. This means that you may be able to trade off the individual worst case cost of each operation for that of the worst case cost over a sequence of operations. In other words, any one particular operation may be slow, but the average time over a sufficiently large number of operations is fast. This is an intuitive definition of amortized time, a way of measuring the complexity of an algorithm. In this case, the algorithms to be concerned with are those that carry out operations on data structures.

The heuristic I'll discuss in this article is called the "structural self-adjusting heuristic." To illustrate what I mean by self-adjusting, consider the following example: Suppose you're running an information warehouse and your taks is to distribute information to people who request it. The information in this warehouse could be stored in a fixed order, such as the order of information in a library. You quickly notice, however, that certain pieces of information are requested more often than others. You could make the job easier by moving the most often

requested information close to the service counter. This means that instead of having to search through the depths of the warehouse at any given time, you have a good portion of the most requested information nearby.

As this example suggests, self-adjusting heuristic algorithms are ideally suited to lists, binary search trees, and priority queues (heaps). In lists, the heuristic attempts to keep the most frequently accessed items as close to the front as possible. In binary search trees, the heuristic attempts to keep the most frequently accessed items as close to the root as possible, while preserving the symmetric ordering. Finally, in heaps, the heuristic attempts to minimize the cost of modifying the structure, and partially orders the heap in a simple, uniform way. To illustrate how these algorithms can be implemented, I've provided sample Pascal source code.

Self-Adjusting Lists

A singly linked list is a group of records where each record contains one field that holds an individual piece of user data, and another field that holds a pointer to the next record in the list. An initial pointer that indicates which record starts the list is (or should be) kept. This pointer enables you to search, insert, and delete operations.

Move-to-Front Singly Linked Lists

To understand how the move-to-front (MTF) approach works, consider a situation in which a particular application uses an open hash table with a linked list that is associated with each array location. Suppose that the hashing routine for this application is as good as it can possibly be. If you wish to improve the search performance without unduly complicating the supporting code, however, you might examine the performance of the search performed on the lists. Chances are that certain elements are accessed more often than others. The use of either a transpose or a frequency count heuristic (two other common access approaches) does not appear to be a good idea because of the search overhead involved with each approach. Both methods require either a local exchange operation or extra searching in order to reinsert an accessed item into the correct part of the list. Also, the count method requires a change in the list: The addition of an integer field that maintains the access count. All three heuristics are effective in that they search less than half the list.

One reason why the MTF heuristic performs better than the transpose method is that the transpose heuristic causes the list to converge more slowly to its optimal ordering. In the case of MTF, an element is brought to the front of the list. Furthermore, such an element quickly "sinks" to the end of the list over the course of a sequence of accesses if that element is not a sufficiently wanted item. Essentially, MTF may be viewed as an optimistic method in the sense that the method "believes" that an accessed item will be accessed again. Analogously, the transpose heuristic may be viewed as pessimistic in that it "doubts" that an accessed item will be accessed again. The count method is a compromise between the two.

As Figure 1 and Listing One (page 105) illustrate, searching is the key operation for the MTF heuristic. This search operation is very much like a normal search on a singly linked list, except that an extra pointer is kept to the current predecessor of the list node that is currently being examined. Once a given item is found, the pointer to its predecessor node is used to alter the predecessor node's link field so that the link field points to the successor node of the accessed item. The link field in the desired node is then altered to point to the first element in the list, and the head-of-list pointer is set to point to the new front-of-the-list item. For all intents and

purposes, the insert operation is a push operation -- the new item is immediately put at the front of the list. Finally, an MTF search is used to perform a delete. If the item to be deleted is located at the front of the list, that item is removed from the front of the list.

Self-Adjusting Heaps

A "heap" is a tree -based data structure in which each record node keeps a key, along with pointers to the record node 's successors. The heap maintains an ordering of items such that every node has a key less than, or equal to, the keys of the node's successors. This last description is the concept of "heap-ordering."

There are a number of classical priority queue structures (such as 2 - 3 trees, leftist heaps, and binomial heaps) that are amenable to fast merging operations. Of these, the simplest scheme for maintaining a heap with fast merge operations is the "leftist heap," which was developed to maintain partially ordered lists with a logarithmic time-merge operation. A leftist heap is based upon a binary tree node that contains rank, weight, and two pointer fields (to indicate left and right children). The rank field is defined to be 0 if a node is a leaf. Otherwise, the rank field is defined as one more than the minimum value of both the rank of the leftchild and the rank of the rightchild.

A binary tree is a leftist heap if it is heap-ordered and if the rank of a given leftchild is greater than, or equal to, the rank of its rightchild sibling. The problem with maintaining leftist heaps is that the configuration of the data structure is based upon the rank definition. All operations are heavily dependent upon the value kept in the rank field of a given node. To illustrate the point, I'll describe the leftist heap merge operation.

The leftist heap merge operation is made possible by a modified version of an "enqueue" process (which takes a heap and a queue pointer's record as parameters). This particular enqueue operation saves the root of the heap and moves the front queue pointer down to the rightchild of the root just saved. You then break the link between the saved root and its rightchild. If the queue pointers are both empty, point the front and rear pointers to the root node that was just saved, and set the rightchild pointer field of the root to empty. Otherwise, point the rightchild pointer to the node currently pointed to by the rear queue pointer, and point the rear queue pointer to this newly obtained node.

Implement the merge with the following steps: While neither of the two heaps being merged is empty, call enqueue with the currently minimum key and with the queue pointer's record. Next, while the "first" heap is not empty, call enqueue with that heap and with the queue pointer's record. Perform the same steps for the other heap. These three processes merge the right path. Complete the process with a bottom-up traversal of the right path in order to fix up the rank fields and to perform any necessary swaps to maintain the structural invariant.

Now point to the current two bottommost nodes on the merge path. If there is no left sibling of the bottommost node, make the rightchild a leftchild and set its parent's rank field to 0. Next, set the rightchild's rightchild pointer field to empty. If the bottommost node has a left sibling, compare the two children and swap them when the rank of the left sibling is less than that of the right sibling. In any event (given this case), set the parent's rank field to 1+ rank of the rightchild. Also note which nodes are the next two bottommost nodes on the merge path at this point, and make sure that the parent node before this step points to the rightchild. This process continues until the root is reached when the root of the new heap is returned. Once the merge

operation for leftist heaps has been described, the other heap operations are easy to implement.

This description of the merge operation suggests that two passes are required over the merge path. The question remains: How do you improve performance without unduly complicating the algorithms that maintain the heap? This can be done with a restructuring method that essentially exchanges every node on the result heap's right path with the node's left sibling. The version of the technique presented here also has a feature in which one top-down pass completes the merge. The resulting structure, called a "top-down-skew heap," is a self-adjusting analog of the leftist heap.

Top-Down-Skew Heaps

A skew heap is based upon a simple binary tree node that contains a weight plus pointer fields to left and right children. The process of merging is made possible by another modified version of the enqueue algorithm. In this case, it's not necessary to maintain the rank/balance field in order to obtain logarithmic, amortized performance.

This particular enqueue operation saves the root of the heap and moves the front queue pointer down to the rightchild of the root just saved. You then break the link between the saved root and its rightchild by changing the current leftchild into a rightchild. If the queue pointers are both empty, point the front and rear pointers to the root node just saved, and set the root's leftchild pointer to empty. Otherwise, the newly obtained node becomes the leftchild of the node that is currently indicated by the rear queue pointer, after which the rear queue pointer is changed to indicate the newly obtained node. (See Figure 2.)

The following steps implement the merge: While neither of the two heaps being merged is empty, call enqueue with the heap that contains the current minimum key and the queue pointer's record. Next, while the "first" heap is not empty, call enqueue with that heap and with the queue pointer's record. Follow the same process for the other heap. (This approach is analogous to Tarjan and Sleator's conceptual noting that the left and right children of every node on the merge path are swapped. The implementation used here, however, is a variation.) Once either of the two heaps being merged becomes empty, merely attach the remaining heap to the bottom of the result heap's left path. Again, the rest of the heap operations are easy to define.

Pairing Heaps

Much like the leftist heap, the binomial heap has an analogous self-adjusting counterpart. This new structure, called the "pairing heap," is a recent development in heaps that supports the decreaseKey operation. The essential definition of the pairing heap, like that of the skew heap, is based upon a simple binary tree record node that contains at least weight plus three pointer fields (to indicate the parent and the left and right siblings). Like most heaps, the pairing heap depends upon a merge operation, but has a less complicated scheme than its classical counterpart.

In the case of the binomial heap, you need to maintain a forest of trees where each tree contains a number of nodes equal to a non-negative integer power of two. Thus, a binomial heap of n items can be represented by a forest in which each tree corresponds one-to-one with the one bit that represents the value of n in binary. (This eventually leads to the fact that all of the binomial heap operations are, in the worst case, logarithmic time.) Needless to say, the code needed to implement a binomial heap merge operation is complicated and difficult to maintain.

The merge operation for pairing heaps begins by determining which of the two heaps has the minimal weight at the root. The heap with the non-minimum key at the root then becomes the

child at the root of the other heap. The heap that is being made into a subtree points its root node right sibling pointer to the child of the root of the other heap. Furthermore, the first heap's parent pointer is set to the new heap root, and the new heap root points its leftchild pointer to the root of the heap that is being made into a subtree. The merge operation returns the root of the new heap. (See Figure 3.)

Given the above definition of the merge operation, the DeleteMin operation (see Figure 4 and Listing Three, page 106) is easy to describe. I will describe the front-back one-pass variation here. To begin, save the root node and keep a pointer to the leftchild. Next, empty the pointer to the root. While subtrees are linked to the leftchild of the root, remove trees in pairs (beginning with the leftchild) and merge the trees, then merge the result to the heap pointed to by the root pointer. Repeat this step until there are no more trees. (The pairing heap derives its name from the restructuring operation that takes place during a DeleteMin.)

Describing the DecreaseKey operation for pairing heaps (see Figure 5) is just as easy. This operation assumes that you have direct access to the node whose weight field is being decreased, to a root to the heap that contains the node, and to the value by which you wish to decrease the weight. Go to the parent of the node that is being operated on, and then go to the leftchild of that parent. Scan along the right sibling list to find the predecessor of the node that will be operated upon. When the predecessor is located, clip out the tree rooted at the node upon which you wish to carry out the actual DecreaseKey operation. To clip out the tree , link around the node in question. If the node is a leftchild, make its right sibling the new leftchild. Now decrease the weight and merge the tree that is rooted at the node with the root of the pairing heap.

The simple local restructuring heuristics presented here provide an elegant approach to the development of heap structures. In fact, these heaps are simpler to understand and to implement than either the leftist or binomial heaps. Furthermore, indications are that self-adjusting heaps are just as competitive in practice as their classical counterparts. In any event, I've presented two very different (though effective) local restructuring heuristics. The first heuristic reorganizes lists in order to make frequently requested list items more accessible. The second heuristic applies a simple local restructuring method (in place of maintaining balance/accounting data and resolving special structural cases) in order to quickly maintain both the structure and the partial ordering of a heap.

Now let's consider an efficient self-adjusting heuristic for binary search trees. This algorithm makes frequently requested items in the tree more easily accessible, and quickly maintains both the structure and the sorted ordering of the tree.

Self-Adjusting Binary Search Trees

In a "binary search tree," each node keeps a key along with two pointers to the node's successors. The ordering is such that if a node has key K, every node in that node's left subtree must have keys less than K, and every node in its right subtree must have keys greater than K. This is known as "symmetric ordering." The performance costs of generic binary search tree operations are, in the worst case, logarithmic time (if the input data is sufficiently random). Such a tree may also degenerate as a result of insertions and deletions, and yield steadily poorer performance.

The process of tree degeneration has led to the development of various height/weight balanced trees and B-tree schemes. Although these various schemes guarantee logarithmic worst

case times per operation, some of the schemes are not as efficient as possible under nonuniform access patterns. Furthermore, many of these schemes require extra space for balance/accounting information, and the need to keep this information current tends to complicate maintenance of the data structure. Certain cases must be checked on each update, thus incurring a large overhead.

Rotation is the key technique that makes some of the balanced and previous self-adjusting tree schemes possible. In fact, rotation plays a part in the implementation of the splay tree. Before this discussion continues, it is necessary to understand how a right rotation and a left rotation at any node of a binary tree are performed.

As Listing Two (page 105) shows, you implement a right rotation with the following steps: If the pointer to a starting root of some tree is not empty and that node has a left subtree, save the pointer to the root and then save the pointer to the right subtree of the initial left subtree. Then make the pointer to the initial left subtree the new starting root pointer, and let the original root be the rightchild of the new root. Finally, designate the pointer to the saved rightchild of the original leftchild as a leftchild of the new root's rightchild (which is the original root).

Implement a left rotation in a similar manner. If the pointer to a starting root of some tree isn't empty, and that node has a right subtree, save the pointer to the root and then save the pointer to the left subtree of the initial right subtree. Then, designate the pointer to the initial right subtree as the new starting root pointer, and let the original root be the leftchild of the new root. Finally, designate the pointer to the saved leftchild of the original rightchild as a rightchild of the new root's leftchild (which is the original root).

The drawbacks of many of the efficient search tree techniques motivated the development of the splay tree. Because binary search trees maintain sorted sets, the question arose as to whether the speed of the search process could be improved if certain items had a higher request frequency than others. In an attempt to improve performance, Allen, Munro, and Bitner proposed two self-adjusting techniques on search trees during the late 1970s. The gist of the first scheme is a single rotation of the item accessed towards the root. The second scheme involves multiple rotations of the accessed item all the way to the root. The techniques are analogous to the variations of the transpose methods for singly linked lists. Neither heuristic is efficient in the amortized sense, since long access sequences exist where the time per access is linear. It is thus clear that the search paths to frequently accessed items need to be as short as possible. Tarjan and Sleator's proposed self-adjusting heuristic halves the depth of each node on the path to an accessed item when the item is finally moved to the root. A splay tree is a binary search tree that employs this heuristic.

Splay Trees

The proposed self-adjusting heuristic has two versions. The "bottom-up splay" is appropriate if you already have direct access to the node that is to be moved to the root. Heuristic restructuring occurs during the second pass back up the search path (assuming that the first pass down the tree is performed in order to find the item). The second version of the proposed self-adjusting heuristic, called a "top-down splay," is an efficient variation of the process used to carry out Tarjan and Sleator's self-adjusting heuristic. This variant requires a pointer to the tree (call it T), that points to the current node to be examined during the search. This heuristic also requires two double pointer records, called L and R (for left and right subtrees), that point to all items less than or greater than the node at T. Figure 6 describes this step.

As illustrated in Figure 7, the splaying process repeatedly applies one of the appropriate cases until there are no more cases to apply. At this point, the leftchild of the remaining root is attached to the bottom right of L, and the rightchild is attached to the bottom left of R. The final step points the leftchild of the final remaining root to the subtrees kept by L, and points the rightchild to the subtrees kept by R. (Unlike the bottom-up variation, the top-down heuristic includes the splay step.) When the search/access for a requested node fails, change the last node on the search path into the root of the tree. This step makes the definition of all of the other operations very easy.

To search for a node, simply apply the searching process as described earlier. The process of insertion involves searching for the key V to be inserted. If V is less than the key at the root, make the node that contains V point its leftchild pointer to the leftchild of the root (which breaks the link from the root to that leftchild), and point the rightchild pointer to the root. Otherwise, the leftchild pointer of the node that contains V points to the root, and the rightchild pointer points to the rightchild of the root (and the link from the root to the rightchild is broken). The insertion step is completed by designating the node that contains V as the root.

The split operation is essentially the process of breaking the tree at the root in the manner described in the description of the insertion process. The process of deletion is just as easy. Perform the splay search from the key to be deleted. If the root does not contain the node with the key to be deleted, nothing happens. If the root does contain the node with the key to be deleted, keep pointers to the two subtrees at the root, perform a splay search for the maximum key in the left subtree (and designate the root of the left subtree as the new root), and point the rightchild pointer of the root to the right subtree. The join operation of two binary search trees (assuming that all items in Tree 1 are less than those in Tree 2) is simply the non-no operation of the delete algorithm just described.

The self-adjusting heuristic provides an alternative to the standard balancing/accounting worst-case asymptotic solutions used to develop efficient programs -- and may, in fact, be the method of choice. Furthermore, the algorithms to maintain these self-adjusting data structures are both conceptually easy to understand and simple to implement in practice.

In which applications could these data structures be used to improve performance? Some possibilities are symbol table management applications (MTF lists, splay trees, and possibly in conjunction with hashing schemes), graph algorithms (skew and pairing heaps, particularly with respect to finding minimum spanning trees and the shortest paths in graphs), and other network optimization algorithms (such as splay trees, particularly in maximum/minimum network flow algorithms). Recent work by Jones, Bern, and de Carvalho, as well as my own work, indicates that some of the self-adjusting data structures do seem to perform better in practice than do conventional data structures.

Bibliography

Aho, A.; Hopcroft, J.; and Ullman, J. The Design and Analysis of Computer Algorithms. Reading, Mass.: Addison-Wesley, 1974.

Allen, B. and Munro, I. "Self-Organizing Search Trees." Journal of the ACM 25 (1978).

Bentley, J.L. and McGeoch, C.C. "Amortized Analyses of Self-Organizing Sequential Search Heuristics." Communications of the ACM 28 (1985).

Bern, M. and de Carvalho, M. "A Greedy Heuristic for the Rectilinear Steiner Tree

Problem." Report No. UCB/CSD 87/306, Computer Science Division. Berkeley: UC Berkeley (1987).

Bitner, J.R. "Heuristics That Dynamically Organize Data Structures." SIAM Journal of Computing 8 (1979).

Brown, M.R. "Implementation And Analysis Of Binomial Queue Algorithms." SIAM Journal of Computing 7 (1978).

Dietz, P. and Sleator, D. "Two Algorithms for Maintaining Order in a List." Proceedings of the 19th ACM Symposium on Theory of Computing (1987).

Fredman, M.L. and Tarjan, R.E. "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms." Proceedings of the 25th Annual IEEE Foundation of Computer Science (1984).

Fredman, M.L. et al. "The Pairing Heap: A New Form of Self-Adjusting Heap." Algorithmica 1 (1986).

Jones, D.W. "An Empirical Comparison of Priority Queue and Event Set Implementations." Communications of the ACM 29 (1986).

Knuth, D.E. The Art Of Computer Programming, Vol. 3: Searching And Sorting, 2nd ed. Reading, Mass.: Addison-Wesley, 1973.

Liao, A.M. "Three Priority Queue Applications Revisited." Submitted to Algorithmica (1988).

Sedgewick, R. Algorithms. Reading, Mass.: Addison-Wesley, 1983.

Sleator, D.D. and Tarjan, R.E. "Self-Adjusting Binary Trees." Proceedings of the 15th ACM Symposium on Theory of Computing (1983).

Sleator, D.D. and Tarjan, R.E. "Self-Adjusting Binary Search Trees." Journal of the ACM 32 (1985).

Sleator, D.D., and Tarjan, R.E. "Self-Adjusting Heaps." SIAM Journal of Computing 15 (1986).

Tarjan, R.E. "Data Structures And Network Algorithms." CBMS Regional Conference Series In Applied Mathematics 44. Philadephia: SIAM (1983).

Tarjan, R.E. "Amortized Computational Complexity." SIAM Journal of Algebraic Discreet Methods 6 (1985).

Vuillemin, J. "A Data Structure For Manipulating Priority Queues." Communications of the ACM 21 (1978).

Wirth, N. Algorithms + Data Structures = Programs. Englewood Cliffs, New Jersey: Prentice Hall, 1976.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to Dr. Dobb's Journal, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the DDJ Forum on CompuServe (type GO DDJ). The DDJ Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

Andrew received his master's degree in computer science from RPI in Troy, New York.

He can be reached through his Bitnet address, which is aliao%eagle @wesleyan bitnet. You can also reach him through the DDJ office.

CAPTIONS: Finding item 12 in a list with the MTF restructuring heuristic. (chart); Merging two skew heaps. (chart); Listing one: singly linked move-to-the front list. (program)